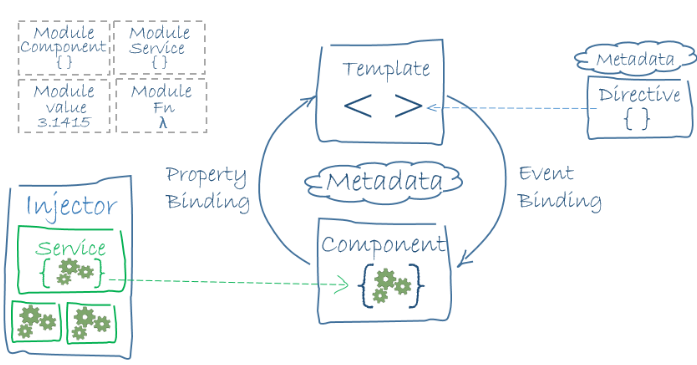
Angular 2

# Introduction

Angular is a framework for building **client applications** in HTML and either JavaScript, or a language (like Dart or TypeScript) that compiles to JavaScript. It is Composed by HTML templates with Angularized markup.

The structure is divided by:

1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Services
8. Dependency injection

## Modules

Every Angular app has at least one module, the root module, conventionally named **AppModule**.

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

1. **Declarations** - the **view classes** that belong to this module. Angular has three kinds of view classes: **components, directives, and pipes**.
2. **Exports** - the subset of **declarations** that should be **visible** **and** **usable** in the component templates of other modules.
3. **Imports** - other modules whose exported **classes are needed** by component templates declared in this module.
4. **Providers** - creators of **services** that this module contributes to the global collection of services; they become **accessible in all parts of the app**.
5. **Bootstrap** - the **main application view**, called the **root component**, that hosts all other app views. Only the root module should set this bootstrap property

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

@NgModule({

imports: [ BrowserModule ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

## Components

A component controls a patch of screen called a view. It defines application logic for a class displayed for the view.

export class HeroListComponent implements OnInit {

heroes: Hero[];

selectedHero: Hero;

constructor(private service: HeroService) { }

ngOnInit() {

this.heroes = this.service.getHeroes();

}

selectHero(hero: Hero) { this.selectedHero = hero; }

}

### Lifecycle hooks

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle by implementing optional lifecycle hooks:

1. **ngOnChanges** - when Angular (re)sets data-bound input properties. Called before ngOnInit and whenever one or more data-bound input properties change.
2. **ngOnInit** - initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called once, after the first ngOnChanges.
3. **ngDoCheck** - Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after ngOnChanges and ngOnInit.
4. **ngAfterContentInit** - Respond after Angular projects external content into the component's view. Called once after the first NgDoCheck.
5. **ngAfterContentChecked** - respond after Angular checks the content projected into the component. Called after the ngAfterContentInit and every subsequent NgDoCheck.
6. **ngAfterViewInit** - respond after Angular initializes the component's views and child views. Called once after the first ngAfterContentChecked.
7. **ngAfterViewChecked** - respond after Angular checks the component's views and child views. Called after the ngAfterViewInit and every subsequent ngAfterContentChecked.
8. **ngOnDestroy** - cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component.

## Templates

You define a component's view with its companion template. A template is a form of **HTML and Angular's template syntax** that tells Angular how to render the component.

Template could contains other component selector, that represents a new component.

<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>

<ul>

<li \*ngFor="let hero of heroes" (click)="selectHero(hero)">

{{hero.name}}

</li>

</ul>

<hero-detail \*ngIf="selectedHero" [hero]="selectedHero"></hero-detail>

## Metadata

Metadata tells Angular how to process a class. In TypeScript, you attach metadata by using a decorator **@Component**.

@Component({

moduleId: module.id,

selector: 'hero-list',

templateUrl: 'hero-list.component.html',

providers: [ HeroService ]

})

export class HeroListComponent implements OnInit {

/\* . . . \*/

}

Here are a few of the possible @Component configuration options:

1. **moduleId**: sets the source of the base address (module.id) for module-relative URLs such as the **templateUrl**.
2. **selector**: **CSS selector** that tells Angular to create and insert an instance of this component where it finds a **tag in parent HTML**.
3. **templateUrl**: module-relative address of this component's **HTML template**, shown above.
4. **providers**: array of **dependency injection providers for services** that the component requires.

## Data binding

Angular supports bidirectional data binding, a mechanism for coordinating parts of a template with parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

<li>{{hero.name}}</li>

<hero-detail [hero]="selectedHero"></hero-detail>

<li (click)="selectHero(hero)"></li>

1. The {{hero.name}} **interpolation** displays the component's hero.name property value within the <li> tags.
2. The [hero] **property binding** passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
3. The (click) **event binding** calls the component's selectHero method when the user clicks a hero's name.

### Two-way data binding

Two-way data binding is an important fourth form that combines property and event binding in a single notation, using the **ngModel** directive. Here's an example from the HeroDetailComponent template:

<input [(ngModel)]="hero.name">

Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.

## Directives

A directive is a class with directive metadata.

Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives.

### Component

In TypeScript, apply the @Directive decorator to attach metadata to the class. A component is a directive-with-a-template; a @Component decorator is actually a @Directive decorator extended with template-oriented features.

### Structural

Structural directives alter layout by adding, removing, and replacing elements in DOM.

<li \*ngFor="let hero of heroes"></li>

<hero-detail \*ngIf="selectedHero"></hero-detail>

[More](#_Structural_Directives)

### Attribute

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The ngModel directive, which implements two-way data binding, is an example of an attribute directive. ngModel modifies the behavior of an existing element (typically an <input>) by setting its display value property and responding to change events.

<input [(ngModel)]="hero.name">

## Services

Service is a broad category encompassing any value, function, or feature that your application needs. Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

There is nothing specifically Angular about services. Angular has no definition of a service. There is no service base class, and no place to register a service.

export class Logger {

log(msg: any) { console.log(msg); }

error(msg: any) { console.error(msg); }

warn(msg: any) { console.warn(msg); }

}

* It is a good practice add @Injectable decorator to a service

@Injectable()

export class HeroService…

### Create Service

1. Name of a service in lower-dash case:

example-hero.service.ts

1. Basic structure:

import { Injectable } from '@angular/core';

@Injectable() export class HeroService { }

* the **@Injectable()** decorator and emits metadata about our service, metadata that Angular may need to inject other dependencies into this service.

1. Declare a function using promise:

getHeroes(): Promise<Hero[]> { return Promise.resolve(HEROES); }

### Injecting Service

For using the previous service:

1. Injecting service
   1. Adding a constructor that also defines a private property

export class AppComponent {

heroes: Hero[];

constructor(private heroService: HeroService) { }

getHeroes(): void {

this.heroService.getHeroes().then(heroes => this.heroes = heroes);

}

}

* 1. Adding the service to the component's providers metadata.

@Component({

providers: [HeroService]

})

* Use constructor always. Don’t create new instance of the service “new Service()…”

1. Using Lifecycle Hook

import { OnInit } from '@angular/core';

export class AppComponent implements OnInit {

ngOnInit(): void { this.getHeroes(); }

}

## Dependency injection

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

constructor(private service: HeroService) { }

You can register providers in modules so the same instance of a service is available everywhere.

providers: [

BackendService,

HeroService,

Logger

],

Also in components, so you get a new instance of the service with each new instance of that component.

@Component({

moduleId: module.id,

selector: 'hero-list',

templateUrl: 'hero-list.component.html',

providers: [ HeroService ]

})

Points to remember about dependency injection:

1. Dependency injection is wired into the Angular framework and used everywhere.
2. The injector is the main mechanism.
   1. An injector maintains a container of service instances that it created.
   2. An injector can create a new service instance from a provider.
3. A provider is a recipe for creating a service.
4. Register providers with injectors.

# Structural Directives

A Structural directive changes the DOM layout by adding and removing DOM elements. We've seen three of the built-in structural directives in other chapters**: ngIf, ngSwitch and ngFor**.

<div \*ngIf="hero">{{hero}}</div>

<div \*ngFor="let hero of heroes">{{hero}}</div>

<div [ngSwitch]="status">

<template [ngSwitchCase]="'in-mission'">In Mission</template>

<template [ngSwitchCase]="'ready'">Ready</template>

<template ngSwitchDefault>Unknown</template>

</div>

## NgIf

It takes a boolean and makes an entire chunk of DOM appear or disappear. **It does not hide** the element, like setting css display style to none, **it removes** the element from DOM, stops change detection for the associated component, detaches it from DOM events (the attachments that it made) and destroys the component. The component can be garbage-collected (we hope) and free up memory.

That could be expensive. There are pros and cons to each approach, in general it is best to use ngIf to remove unwanted components rather than hide them.

## The <template> tag

Structural directives, like ngIf, do their magic by using the **HTML 5 template tag**. Angular replaces the <template> tag and its contents with empty <script> tags. That's just its default behavior. It can do something different as we saw when applying a variety of ngSwitch directives to <template> tags. See the next section.

## NgSwitch

When one of those ngSwitch conditions is true, Angular inserts the template's content into the DOM.

<div [ngSwitch]="status">

<template [ngSwitchCase]="'in-mission'">In Mission</template>

<template [ngSwitchCase]="'ready'">Ready</template>

<template ngSwitchDefault>Unknown</template>

</div>

## The asterisk (\*) effect

The asterisk is "syntactic sugar". It simplifies ngIf and ngFor for both the writer and the reader. Under the hood, Angular replaces the asterisk version with a more verbose <template> form.

<!-- Examples (A) and (B) are the same -->

<!-- (A) \*ngIf paragraph -->

<p \*ngIf="condition">

Our heroes are true!

</p>

<!-- (B) [ngIf] with template -->

<template [ngIf]="condition">

<p>

Our heroes are true!

</p>

</template>

## Make a structural directive

Creating a directive is similar to creating a component.

1. import the Directive decorator.
2. add a CSS attribute selector (in brackets) that identifies our directive.
3. specify the name of the public input property for binding (typically the name of the directive itself).
4. apply the decorator to our implementation class.

import { Directive, Input } from '@angular/core';

import { TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[myUnless]' })

export class UnlessDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef

) { }

@Input() set myUnless(condition: boolean) {

if (!condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear();

}

}

}

# Form module

Framework support for two-way data binding, change tracking, validation, and error handling...

## App module

import { FormsModule } from '@angular/forms';

@NgModule({

imports: [FormsModule]

})

## Example

<div class="form-group">

<label for="power">Hero Power</label>

<select class="form-control" id="power"

required

[(ngModel)]="model.power" name="power">

<option \*ngFor="let p of powers" [value]="p">{{p}}</option>

</select>

</div>

## Get user input from a template reference variable

These variables grant us direct access to an element. We declare a template reference variable by preceding an identifier with a hash/pound character (#).

@Component({

selector: 'key-up2',

template: `

<input #box (keyup)="onKey(box.value)">

<p>{{values}}</p>

`

})

export class KeyUpComponent\_v2 {

values = '';

onKey(value: string) {

this.values += value + ' | ';

}

}

## Key event filtering

Maybe we're only interested in the input box value when the user presses Enter, and we'd like to ignore all other keys.

@Component({

selector: 'key-up3',

template: `

<input #box (keyup.enter)="values=box.value">

<p>{{values}}</p>

`

})

export class KeyUpComponent\_v3 {

values = '';

}

## On blur Event

Listening to the input box's blur event as well

@Component({

selector: 'key-up4',

template: `

<input #box

(keyup.enter)="values=box.value"

(blur)="values=box.value">

<p>{{values}}</p>

`

})

export class KeyUpComponent\_v4 {

values = '';

}

# NgFor

Built-in directive \*ngFor is used for looping over array. For example, we loop over heroes:

<li \*ngFor="let hero of heroes">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

The leading asterisk (\*) in front of ngFor is a critical part of this syntax.